Scalable Performance Tuning of Hadoop MapReduce: A Noisy Gradient Approach

Sandeep Kumar*, Sindhu Padakandla*, Chandrashekar L†, Priyank Parihar, Gopinath K* and Shalabh Bhatnagar*
* *Dept. of Computer Science and Automation*
Indian Institute of Science
{*sandeepkumariisc, sindhupr*}*@gmail.com*
{*gopi, shalabh*}*@csa.iisc.ernet.in*
† *Department of Computing Science*
University of Alberta
chandrurec5@gmail.com

***Abstract*—Hadoop MapReduce is a popular framework for distributed storage and processing of large datasets and is used for big data analytics. It has various configuration parameters which play an important role in deciding the performance i.e., the execution time of a given big data processing job. Default values of these parameters do not result in good performance and therefore it is important to tune them. However, there is inherent difficulty in tuning the parameters due to two important reasons - first, the parameter search space is large and second, there are cross-parameter interactions. Hence, there is a need for a dimensionality-free method which can automatically tune the configuration parameters by taking into account the cross-parameter dependencies. In this paper, we propose a novel Hadoop parameter tuning methodology, based on a noisy gradient algorithm known as the simultaneous perturbation stochastic approximation (SPSA). The SPSA algorithm tunes the selected parameters by directly observing the performance of the Hadoop MapReduce system. The approach followed is independent of parameter dimensions and requires only 2 observations per iteration while tuning. We demonstrate the effectiveness of our methodology in achieving good performance on popular Hadoop benchmarks namely *Grep*, *Bigram*, *Inverted Index*, *Word Co-occurrence* and *Terasort*. Our method, when tested on a 25 node Hadoop cluster shows 45-66% decrease in execution time of Hadoop jobs on an average, when compared to prior methods. Further, our experiments also indicate that the parameters tuned by our method are resilient to changes in number of cluster nodes, which makes our method suitable to optimize Hadoop when it is provided as a service on the cloud.**

***Keywords*-Hadoop Parameter Tuning, Simultaneous Perturbation Stochastic Approximation, Cloud Computing**

I. Introduction

We are in the era of big data and huge volumes of data are generated in various domains like social media, financial markets, transportation etc. Quick analysis of such huge quantities of unstructured data is a key requirement for achieving success in many of these domains. Performing distributed sorting, extracting hidden patterns and unknown correlations and other useful information is critical for making better decisions. To efficiently analyze large volumes of data, there is a need for parallel and distributed processing/programming methodologies.

MapReduce [1] is a popular computation framework which is optimized to process large amounts of data. It is designed to process data in parallel and in a distributed fashion using resources built out of commodity hardware. MapReduce deviates from the norm of other computation frameworks as it minimizes the movement of data. The *Map* and *Reduce* phases analyze the data which is split into several chunks and stored in a distributed manner across nodes. The main operational logic of MapReduce is based on $< key, value >$ pairs. All the operations are done on these key value pairs. Apache **Hadoop** [2] is a popular open-source implementation of MapReduce. Besides MapReduce, it comprises of the Hadoop Distributed File System (HDFS), which stores data and manages the Hadoop cluster (see [2]). Hadoop cluster consists of a master node which handles the scheduling of jobs and placement of files on the cluster. The rest of the nodes are slave nodes where the actual execution of job and storing of file is done.

The Hadoop framework provides different parameters that can be tuned according to the workload and hardware resources. For e.g., a file is split into one or more data blocks and these blocks are stored in a set of DataNodes. The block size is controlled by a parameter *dfs.block.size*, which can be set based on the input data size of the workload and the cluster configuration. The performance of an application running on the Hadoop framework is affected by the values of such parameters. The default values of these parameters generally do not give a satisfactory performance. Therefore, it is important to tune these parameters according to the workload. The performance of an application on Hadoop cannot be quantified in terms of these parameters and hence finding best parameter value configuration for a given application proves to be a tricky task. In addition, it is difficult to tune these parameters owing to two other reasons. First, due to the presence of a large number of parameters (about 200, encompassing a variety of functionalities) the search space is large and complex. Second, there is a pronounced effect of cross-parameter interactions, i.e., the parameters are not independent of each other. For instance, the parameter *io.sort.mb* controls the number of spills written to disk (map phase). If it is set high, the spill percentage of Map (controlled by *sort.spill.percent*) should also be set to a high value. Thus, the complex search space along with the cross-parameter interaction does not make Hadoop amenable to manual tuning.

Need for tuning the Hadoop parameters to enhance the

2159-6190/17 $31.00 © 2017 IEEE
DOI 10.1109/CLOUD.2017.55

performance was identified in [3]. Attempts toward building an optimizer for hadoop performance started with Starfish [4]. Recent efforts in the direction of automatic tuning of the Hadoop parameters include MROnline [5] and PPABS [6]. We observe that collecting statistical data to create virtual profiles and estimating execution time using mathematical model (as in [3]-[6]) requires significant level of expertise. Moreover, since Hadoop MapReduce is evolving continuously with changes in management of workloads, the mathematical model also has to be updated. With new versions of Hadoop being released, these mathematical models might not be applicable, due to which a model-based approach might fail.

In order to address the above shortcomings, we suggest a method that directly utilizes the data from the real system and tunes the parameters via *feedback*. This approach is based on a noisy gradient stochastic optimization method known as the simultaneous perturbation stochastic approximation (SPSA) algorithm [7]. In our work, we adapt the SPSA algorithm to tune the parameters used by Hadoop to allocate resources for program execution.

Our paper is organised as follows: we provide a detailed description of our SPSA-based approach in the next section. Following it, we describe the experimental setup and present the results in Section III. Section IV concludes the paper and suggests future enhancements.

II. Automatic Parameter Tuning

The performance of various complex systems such as traffic control, unmanned aerial vehicle (UAV) control etc. depends on a set of tunable parameters (denoted by θ). Parameter tuning in such cases is difficult because of the *black-box* nature of the problem and the *curse-of-dimensionality* [8]. In this section, we discuss the general theme behind the methods that tackle these bottlenecks and their relevance to the problem of tuning the Hadoop parameters.

A. Bottlenecks in Parameter Tuning

In many systems, the exact nature of the dependence of the performance on the parameters is not known explicitly i.e., the performance of the system cannot be expressed as an analytical function of the parameters. As a result, the parameter setting that offers the best performance cannot be computed apriori. However, the performance of the system can be observed for any given parameter setting either from the system or a simulator of the system. Hadoop MapReduce exhibits this black-box nature, because it is not well structured like SQL. In such systems, one can resort to *black-box* or *simulation-based* optimization methods that tune the parameters based on the output observed from the system/simulator without knowing its internal functioning. As illustrated in Fig. 1, the black-box optimization scheme sets the current value of the parameter based on the past observations.

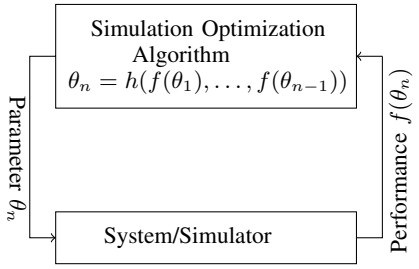


Figure 1: The simulation optimization algorithm makes use of the *feedback* received from the system/simulator to tune the parameters. Here $n = 1, 2, \ldots$ denotes the trial number, θ_n is the parameter setting at the n^{th} trial and $f(\cdot)$ is the performance measure. The map h makes use of past observations to compute the current parameter setting.

The following issues arise in the context of black-box optimization:

1) Number of observations made and the cost of obtaining an observation from the system/simulator - These are directly dependent on the size of the parameter space. As the number of parameters increase, there will be an exponential increase in the size of the search space, which is often referred to as the curse-of-dimensionality. Additionally, in many applications, even though the number of parameters are small, the search space will still be huge because each of the parameters can take continuous values, i.e., values in R. Since it is computationally expensive to search such a large parameter space, it is important for black- box optimization methods to make as few observations as possible.
2) Cross-parameter dependencies - Parameters cannot be assumed to be independent of each other. A black-box optimization method must have some technique to take into account cross-parameter interactions and still be able to provide a set of optimal parameters.

B. Noisy Gradient based Optimization

In order to take the cross-parameter interactions into account, one has to make use of the sensitivity of the performance measure with respect to each of the parameters at a given parameter setting. This sensitivity is formally known as the *gradient* of the performance measure at a given setting. If there are n parameters to tune, then it takes only $O(n)$ observations to compute the gradient of a function at a given point. However, even $O(n)$ computations are not desirable if each observation is itself costly.

Consider the noisy gradient scheme given in (1) below.

$$\theta_{n+1} = \theta_n - \alpha_n\big(\nabla f_n + M_n\big), \tag{1}$$

where $n = 1, 2 \ldots$ denotes the iteration number, $\nabla f_n \in \mathbb{R}^n$ is the gradient of function f, $M_n \in \mathbb{R}^n$ is a zero-mean noise

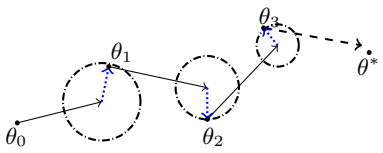


Figure 2: Noisy Gradient scheme. Notice that the noise can be filtered by an appropriate choice of diminishing step sizes.

sequence and α_n is the step-size. Fig. 2 presents an intuitive picture of how a noisy gradient algorithm works. Here, the algorithm starts at θ_0 and needs to move to θ^* which is the desired optimal solution. The solid lines denote the true gradient step (i.e., $\alpha_n \nabla f_n$) and the dash-dotted circles show the region of uncertainty due to the noise term $\alpha_n M_n$. The dotted line denotes the fact that the true gradient is disturbed and each iterate is *pushed* to a different point within the region of uncertainty. The idea here is to use diminishing step-sizes to filter the noise and eventually move towards θ^*. The simultaneous perturbation stochastic approximation (SPSA) algorithm is a *noisy gradient* algorithm which works as illustrated in Fig. 2. It requires only 2 observations from the system per iteration. Thus the SPSA algorithm is extremely useful in cases when the dimensionality is high and the observations are costly. We adapt it to tune the parameters of Hadoop. By adaptively tuning the Hadoop parameters, we intend to optimize the Hadoop job execution time, which is the performance metric (i.e., $f(\theta)$) in our experiments.

C. Simultaneous Perturbation Stochastic Approximation (SPSA)

We use the following notation:

1) $\theta \in X \subset \mathbb{R}^n$ denotes the tunable parameter. Here n is the dimension of the parameter space. Also, X is assumed to be a compact and convex subset of $\mathbb{R}^n$.
2) Let $x \in \mathbb{R}^n$ be any vector, then $x(i)$ denotes its i^{th} co-ordinate, i.e., $x = (x(1), \ldots, x(n))$.
3) $f(\theta)$ denotes the performance of the system for parameter θ. Let f be a smooth and differentiable function of θ.
4) $\nabla f(\theta) = (\frac{\partial f}{\partial \theta(1)}, \ldots, \frac{\partial f}{\partial \theta(n)})$ is the gradient of the function, and $\frac{\partial f}{\partial \theta(i)}$ is the partial derivative of f with respect to $\theta(i)$.
5) $e_i \in \mathbb{R}^n$ is the standard n-dimensional unit vector with 1 in the i^{th} co-ordinate and 0 elsewhere.

Formally the gradient is given by

$$\frac{\partial f}{\partial \theta(i)} = \lim_{h \to 0} \frac{f(\theta + h e_i) - f(\theta)}{h}. \tag{2}$$

In (2), the i^{th} partial derivative is obtained by perturbing the i^{th} co-ordinate of the parameter alone and keeping rest of the co-ordinates the same. Thus, we require $n+1$ operations to compute the gradient once using perturbations. This can be a shortcoming in cases when it is computationally expensive to obtain measurements of f and the number of parameters is large.

The idea behind the SPSA algorithm is to perturb not just one co-ordinate at a time but all the co-ordinates together simultaneously in a random fashion. However, one has to carefully choose these random perturbations so as to be able to compute the gradient. Formally, a random perturbation $\Delta \in \mathbb{R}^n$ should satisfy the following assumption.

Assumption 1: For any $i \neq j, i = 1, \ldots, n, j = 1, \ldots, n$, the random variables $\Delta(i)$ and $\Delta(j)$ are zero-mean,independent, and the random variable Z_{ij} given by $Z_{ij} = \frac{\Delta(i)}{\Delta(j)}$ is such that $\mathbf{E}[Z_{ij}] = 0$ and it has finite second moment.

An example of such a random perturbation is the following: Let $\Delta \in \mathbb{R}^n$ be such that, each of its co-ordinates $\Delta(i)$ is an independent Bernoulli random variable taking values -1 or $+1$ with equal probability, i.e., $Pr\{\Delta(i) = 1\} = Pr\{\Delta(i) = -1\} = \frac{1}{2}$ for all $i = 1, \ldots, n$. This random variable satisfies Assumption 1.

D. Noisy Gradient Recovery from Random Perturbations

Let $\hat{\nabla} f_\theta$ denote the gradient estimate, and let $\Delta \in \mathbb{R}^n$ be any perturbation vector satisfying Assumption 1. Then for any small positive constant $\delta > 0$, the one-sided SPSA algorithm [9] obtains an estimate of the gradient according to equation (3) given below.

$$\hat{\nabla} f_\theta(i) = \frac{f(\theta + \delta\Delta) - f(\theta)}{\delta \Delta(i)}. \tag{3}$$

We now look at the *expected* value of $\hat{\nabla} f_\theta(i)$, which is given by the following:

$$\mathbf{E}[\hat{\nabla} f_\theta(i) | \theta] = \frac{\partial f}{\partial \theta(i)} + o(\delta). \tag{4}$$

The above equation can be easily computed, since $\mathbf{E}\left[\frac{\partial f}{\partial \theta(j)} \frac{\Delta(j)}{\Delta(i)} | \theta\right] = 0$. This follows from the property of Δ in Assumption 1. Thus $\mathbf{E}[\hat{\nabla} f_\theta(i)] \to \nabla f_\theta(i)$ as $\delta \to 0$. Notice that in order to compute the gradient ∇f_θ at the point θ the SPSA algorithm requires only 2 measurements namely $f(\theta - \delta\Delta)$ and $f(\theta + \delta\Delta)$. An extremely useful consequence is that the gradient estimate is not affected by the number of dimensions. The complete SPSA algorithm is shown in Algorithm 1, where $\{\alpha_n\}$ is the step-size schedule and Γ is a projection operator that keeps the iterates within X.

Algorithm 1 uses a noisy gradient estimate (in line 6) and at each iteration takes a step in the negative gradient direction so as to minimize the cost function. The noisy gradient update can be re-written as

$$\begin{aligned} \theta_{n+1} &= \Gamma\left(\theta_n - \alpha_n\big(\mathbf{E}[\hat{\nabla} f_n | \theta_n] + \hat{\nabla} f_n - \mathbf{E}[\hat{\nabla} f_n | \theta_n]\big)\right) \\ &= \Gamma\left(\theta_n - \alpha_n\big(\nabla f_n + M_{n+1} + \epsilon_n\big)\right) \end{aligned} \tag{5}$$

Algorithm 1 Simultaneous Perturbation Stochastic Approximation

Let initial parameter setting be $\theta_0 \in X \subset \mathbb{R}^n$
for $n = 1, 2 \ldots, N$ **do**
  Observe the performance of system $f(\theta_n)$.
  Generate a random perturbation vector $\Delta_n \in \mathbb{R}^n$.
  Observe the performance of system $f(\theta_n + \delta\Delta_n)$.
  Compute the gradient estimate $\hat{\nabla} f_n(i) = \frac{f(\theta_n+\delta\Delta_n)-f(\theta_n-\delta\Delta_n)}{2\times\delta\Delta_n(i)}$.
  Update the parameter in the negative gradient direction $\theta_{n+1}(i) = \Gamma\big(\theta_n(i) - \alpha_n \frac{f(\theta_n+\delta\Delta_n)-f(\theta_n)}{\delta\Delta_n(i)}\big)$.
end for
return θ_{N+1}

where $M_{n+1} = \hat{\nabla} f_n - \mathbf{E}[\hat{\nabla} f_n | \theta_n]$ is an associated martingale difference sequence under the sequence of σ-fields $\mathfrak{F}_n = \sigma(\theta_m, m \leq n, \Delta_m, m < n),\ n \geq 1$ and ϵ_n is a small bias due to the $o(\delta)$ term in (4). The iterative update in (5) is known as a stochastic approximation [10] recursion. As per the theory of stochastic approximation, in order to *filter* out the noise, the step-size schedule $\{\alpha_n\}$ needs to satisfy the conditions below.

$$\sum_{n=0}^{\infty} \alpha_n = \infty, \sum_{n=0}^{\infty} \alpha_n^2 < \infty. \tag{6}$$

The first of the conditions in (6) ensures that the algorithm does not converge to a non-optimal parameter configuration prematurely, while the second ensures that the noise asymptotically vanishes. If the above conditions are satisfied, the iterates of the algorithm converge to local minima (see [10]). However, in practice local minima corresponding to small valleys are avoided due either to the noise inherent in the update or one can periodically inject some noise so as to let the algorithm explore further. Also, though the result stated in [10] is only asymptotic in nature, in most practical cases convergence is observed in a finite number of steps.

E. Adapting SPSA Algorithm to tune Hadoop Parameters

The performance of an application running on Hadoop can be measured by different metrics - namely amount of memory used, number of jobs spawned, execution time etc. Out of these, measuring the execution time of a job is the most practical, as it gives an idea about the job dynamics. Our method involves adapting SPSA algorithm to improve the execution time of jobs running on Hadoop/Hive. Thus, with respect to Section II B, the function $f(\theta)$ hereon denotes the execution time of a workload for a given parameter vector θ. We believe that this multi-variate function representing the execution time of a workload is multimodal and has multiple local minima (we do not assume any form of the function - like convexity, or non-linear). Thus, finding a local minima is as good as finding a global minimizer. All minima of the function lead to a similar performance with respect to the execution time.

The SPSA algorithm needs each of the parameter components to be real-valued i.e., $\theta \in X \subset \mathbb{R}^n$. However, most of the Hadoop parameters that are of interest are not $\mathbb{R}^n$-valued. Thus, on the one hand we need a set of $\mathbb{R}^n$-valued parameters that the SPSA algorithm can tune and a mapping that takes these $\mathbb{R}^n$-valued parameters to the Hadoop parameters. In order to make things clear we introduce the following notation:

1) The Hadoop parameters are denoted by θ_H and the $\mathbb{R}^n$-valued parameters tuned by SPSA are denoted by θ_A[1].
2) S_i denotes the set of values that the i^{th} Hadoop parameter can assume. $\theta_H^{\min}(i)$, $\theta_H^{\max}(i)$ and $\theta_H^d(i)$ denote the *minimum*, *maximum* and *default* values that the i^{th} Hadoop parameter can assume.
3) $\theta_A \in X \subset \mathbb{R}^n$ and $\theta_H \in S_1 \times \ldots \times S_n$.
4) $\theta_H = \mu(\theta_A)$, where μ is the function that maps $\theta_A \in X \subset \mathbb{R}^n$ to $\theta_H \in S_1 \times \ldots \times S_n$.

In this paper, we choose $X = [0,1]^n$, and $\mu : \theta \to \mathbb{R}^n$ such that $\mu(\theta_A)(i) = y$, where

$$y = (\theta_H^{\max}(i) - \theta_H^{\min}(i))\theta_A(i) + \theta_H^{\min}(i)$$

. For an integer valued parameter, we let $\mu(\theta_A)(i) = \lfloor y \rfloor$. We chose $\delta\Delta_n \in \mathbb{R}^n$ to be independent random variables, such that $Pr\{\delta\Delta_n(i) = -\frac{1}{\theta_H^{\max}(i)-\theta_H^{\min}(i)}\} = Pr\{\delta\Delta_n(i) = +\frac{1}{\theta_H^{\max}(i)-\theta_H^{\min}(i)}\} = \frac{1}{2}$. This perturbation sequence ensures that the Hadoop parameters assuming only integer values change by a magnitude of at least 1 in every perturbation. Otherwise, using a perturbation whose magnitude is less than $\frac{1}{\theta_H^{\max}(i)-\theta_H^{\min}(i)}$ might not cause any change to the corresponding Hadoop parameter resulting in an incorrect gradient estimate.

The conditions for the step-size schedule $\{\alpha_n\}$ are asymptotic in nature and are required to hold for convergence to local minima. However, in practice, a constant step size can be used since one reaches closer to the desired value in a finite number of iterations. We know apriori that the parameters tuned by the SPSA algorithm belong to the interval $[0,1]$ and it is enough to have step-sizes of the order of $\min_i(\frac{1}{\theta_H^{\max}(i)-\theta_H^{\min}(i)})$ (since any finer step-size used to update the SPSA parameter $\theta_A(i)$ will not cause a change in the corresponding Hadoop parameter $\theta_H(i)$). In our experiments, we chose $\alpha_n = 0.01, \forall n \geq 0$ and observed convergence in about 20 iterations.

III. Experimental Evaluation

In this paper, we have used use Hadoop versions 1.0.3, 2.7.3 and Hive version 2.1.1 for our experiments. First, we justify the selection of parameters to be tuned in our

[1]Here subscripts A and H are abbreviations of the keywords Algorithm and Hadoop respectively

experiments. Then, we give details about the implementation followed by discussion of results.

A. Parameter Selection

Hadoop consists of myriad of tunable parameters, however most of them are concerned with Hadoop setup and bookkeeping activities and does not affect the performance of workloads. Based on the data flow analysis of Hadoop MapReduce (see [2]) we identify 11 parameters which are found to critically affect the operation of HDFS and the Map/Reduce operations (listed in Table I). We tune parameters which are directly Hadoop dependent, for e.g., number of reducers, I/O utilization parameter etc. and avoid tuning parameters, which are not directly related to Hadoop such as *mapred.child.java.opts* which are best left for cluster or OS level optimization.

B. Cluster Setup and Benchmarks

Our Hadoop cluster consists of 25 nodes. Each node has a 8 core Intel Xeon E3, 2.50 GHz processor, 3.5 TB HDD, 16 GB memory and Gigabit connectivity.

In order to evaluate the performance of our method, we use representative benchmark applications. *Terasort* takes as input a text data file (generated using *teragen*) and sorts it. *Grep* searches for a particular pattern in a given input file. *Bigram* counts all unique sets of two consecutive words in a set of documents, while *Inverted Index* generates word to document indexing from a list of documents. *Word Co-occurrence* is a popular Natural Language Processing program which computes the word co-occurrence matrix of a large text collection.

C. Hive Workload

MapReduce, although capable of processing huge amounts of data, is often used with a wrapper around it to ease its usage. Hive [11] is one such wrapper which provides an environment where MapReduce jobs can be specified using SQL. Its performance can be tested on clusters using the TPCDS benchmark. This benchmark generates data (scale factor) as per requirement and executes a suite of SQL queries on the generated data. The SQL queries focus on different aspects of performance. In our experiments, we have selected 4 queries, which are first optimized using SPSA (see Fig. 3). Following this, we compare the performance of TPCDS using the parameters tuned by SPSA algorithm and values suggested in the TPCDS manual (see Fig. 4d).

D. Learning/Optimization Phase

SPSA runs a Hadoop job with a different configuration in each iteration. We refer to these iterations as the *optimization* or the *learning* phase. The algorithm eventually converges to an optimal value of the configuration parameters. The job execution time corresponding to the converged parameter vector is optimal for the corresponding application. During our evaluations we have seen that SPSA algorithm converges within 10 - 15 iterations and within each iteration it makes two observations, i.e. it executes Hadoop job twice, taking the total count of Hadoop runs during the optimization phase to 20 - 30. It is of utmost importance that the optimization phase is fast, otherwise it can overshadow the benefits which it provides.

Partial Data: In order to ensure a fast optimization phase, we execute the Hadoop jobs (during this phase) on a *partial* workload. Deciding the size of this *partial* workload (denoted N_p) is crucial as the run time on a small work load will be eclipsed by the job setup and cleanup time. We take cue from Hadoop processing system to determine the size of the partial workload. Hadoop splits the input data based on the block size (denoted b_S) of HDFS and spawns a map for each of the splits. The number of map tasks that can run in parallel at a given time is upper bounded by the total map slots (denoted m) available in the cluster. Using this fact, we set the size of the partial data set as:

$$N_p = 2 * b_S * m$$

Using workload of this size, Hadoop will use two waves of the maps jobs to finish the map operation. That will allow SPSA algorithm to capture the statistics of a single wave and the correlations between two successive waves. Our expectation (and as borne by our results) is that the value of configuration parameters which optimize these two waves of map jobs also optimize all the subsequent waves as those are repetitions of similar map jobs. In the cases of *Bigram* and *Inverted Index* benchmark, we observed that even with small amount of data, the job take a long time finish, as they are *reduce-intensive* benchmarks. So, while training these benchmarks, we have used small sized input data files, which results in absence of two waves of map tasks. However, since in these applications, reduce operations take precedence, the absence of two waves of map tasks did not create much of a hurdle.

E. Experimental Setup

We optimize *Terasort* using a partial data set of size $30GB$, *Grep* on $22GB$, *Word Co-occurrence* and *Inverted Index* on 1GB and *Bigram* count on $200MB$ of data set. For Word-Cooccurrence, *Grep* and *Bigram* benchmarks we use the Wikipedia dataset of $\approx 50GB$ [5]. We use the *default* configuration of the parameters as the initial point for the optimization during the learning/optimization phase.

SPSA algorithm terminates when either the change in gradient estimate is negligible or the maximum number of iterations have been reached. An important point to note is that Hadoop parameters can take values only in a fixed range. We handle this by *projecting* the tuned parameter values into the range set (component-wise).

F. Discussion of Results

We compare our method with *Starfish* [4] and *Profiling and Performance Analysis-based System* (PPABS) [6] frameworks. Starfish is designed for Hadoop v1 only, whereas PPABS works with the recent versions also. To run Starfish, we use the executable hosted by the authors of [4] to profile the jobs run on partial workloads. Then execution time of new jobs is obtained by running the jobs using parameters provided by Starfish. For testing PPABS, we collect and use the datasets as described in [6].

1) Performance Evaluation: Our method initializes the parameters to the default values. The execution time corresponding to this initial parameter configuration is given by first data point in all plots of Fig. 3. The initial variation in the execution times of the benchmarks is due to the noisy nature of the gradient estimate. These eventually die down after a few iterations. Since the execution times of the applications are stochastic in nature, we run 10 Monte Carlo simulations with the optimal parameters tuned by SPSA. This gives us the average execution time as well as the standard deviation for each benchmark.

As can be observed from Fig. 4, SPSA reduces the execution time of *Terasort* benchmark by $60\% - 63\%$ when compared to default settings and by $40\% - 60\%$ when compared to Starfish optimizer. For *Inverted Index* benchmark the reduction is about 80% when compared to default settings and about 40% when compared to Starfish. In the case of *Word Co-occurrence*, the observed reduction is 70% when compared to default settings and 50% when compared to Starfish.

SPSA, while finding the optimal configuration, factors in the co-relation among the parameters (Table I). For example, in *Terasort*, a small value (0.14 or 14%) of *io.sort.spill.percent* will generate a lot of spilled files of small size. Because of this, the value of *io.sort.factor* has been increased to 475 from the default value of 10. This will ensure that a large number of spilled files (475) are combined to generate the partitioned and sorted file. Similarly, *shuffle.input.buffer.percent* and *inmem.merge.threshold* (0.14 and 9513 respectively) act as a threshold beyond which in-memory merge of files (output by map) is triggered. Because of low value of *io.sort.spill.percent* the size of the spilled files will be small, and it would take a large number of files (9513) to fill the 14% of reduce memory.

Default value of number of reducers (i.e., 1) generally does not work in practical situations. However, increasing it to a very high number also creates an issue as it results in more network and disk overhead. SPSA optimizes this based on the map output size. *Grep* benchmark, produces very little map output, and even smaller sized data to be shuffled. Hence *io.sort.mb* value is reduced to 50 from default 100 (see Table I) and number of reducers is set to 1. Further, value of *inmem.merge.threshold* has been reduced to 681 from 1000 as there is not much data to work on.

The difference in the values of the tuned parameter in Hadoop 1 and Hadoop 2 for the same benchmark arises because of inherent differences in their architecture and how jobs are executed.

2) Dynamically Changing Cluster Configuration: The tuning of parameters by SPSA is independent of the number of nodes in the cluster. This is substantiated by the results shown in Fig. 4c, which shows the execution times of benchmark applications when the number of nodes in the cluster was reduced to half. The parameter values provided by SPSA are same as the optimal values provided by it, for the original cluster configuration (as described in Section III-B). The execution times (in seconds) for the benchmark applications, based on multiple Monte Carlo simulations are: Terasort: 456 ± 13.12, Bigram: 630 ± 17, Inverted Index: 251 ± 4.2 and Word Co-occurrence: 2786.4 ± 16.9.

The tuning of parameters by SPSA depends on the configuration of the nodes (which is not changed when nodes are added or removed), as some Hadoop parameters like io.sort.mb are influenced by memory available in each node. The lack of dependency on number of nodes in the cluster comes from the fact that SPSA optimizes the performance of a single map or reduce wave. This results in optimization of all subsequent waves. Addition or removal of nodes means there will be less or more number of map-reduce waves.

3) Comparison against Random Configuration: As a sanity check, we compare the performance of SPSA against a random setting of the configuration parameters. When compared with default setting, where the number of reducer is 1, one can get a better performance by just increasing this number. However, that is not optimal and can be improved further by using a more mathematical proven techniques like SPSA. The results (average of 5 executions) are shown in figure 4c. One issue with random setting is that sometime it will lead to failure of job. We have not considered those jobs for the average result calculation.

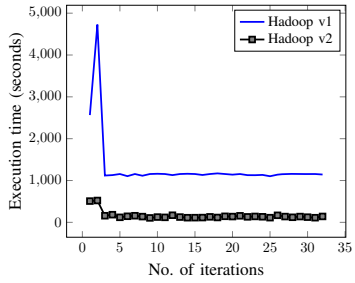
G. Advantages of SPSA

The above discussion indicates that SPSA performs well in optimizing Hadoop parameters. Below, we summarize the advantages of using our proposed method:

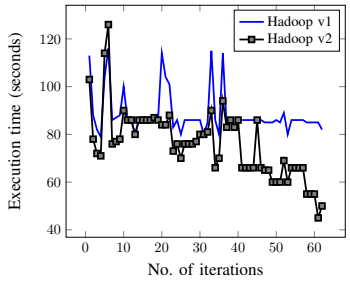
1) Scalable: Parameters can be easily added and removed from the set of tunable parameters, which make our method suitable for scenarios where the user wants to have control over the parameters to be tuned.
2) Independent of Hadoop version: SPSA does not rely on the internal structure of Hadoop and only observes the final execution time (easily accessible).
3) SPSA takes into consideration multiple values of execution time of a job for the same parameter setting (randomness in execution time). Multiple observations helps SPSA to remove the randomness in the job which arise due to the underlying hardware.

Parameter Name	Default	Terasort		Grep		Bigram		Inverted Index		Word Co-occurrence	
		v1.0.3	v2.7.3	v1.0.3	v2.7.3	v1.0.3	v2.7.3	v1.0.3	v2.7.3	v1.0.3	v2.7.3
io.sort.mb	100	149	**524**	50	291	**751**	779	872	202	221	912
io.sort.spill.percent	0.80	**0.14**	0.89	0.83	0.88	0.53	0.53	0.2	0.68	0.75	0.47
io.sort.factor	10	**475**	115	5	57	5	178	50	85	40	5
shuffle.input.buffer.percent	0.7	0.86	0.87	0.67	0.78	0.43	0.43	0.83	0.58	0.65	**0.37**
shuffle.merge.percent	0.66	**0.14**	0.83	0.63	0.74	0.89	**0.39**	0.83	0.54	0.71	0.33
inmem.merge.threshold	1000	**9513**	318	**681**	200	**4201**	200	1095	948	1466	**200**
reduce.input.buffer.percent	0.0	0.14	0.19	0.13	0.0	0.31	0.0	0.17	0.07	0.12	0.0
mapred.reduce.tasks	1	**95**	22	1	1	**33**	35	**76**	16	14	**41**
io.sort.record.percent	0.05	0.14	-	0.1	-	0.31	-	0.17	-	0.2	-
mapred.compress.map.output	false	true	-	false	-	false	-	true	-	false	-
mapred.output.compress	false	false	-	false	-	false	-	false	-	false	-
reduce.slowstart.completedmaps	0.05	-	0.23	-	0.13	-	0.05	-	0.18	-	0.4
mapreduce.job.jvm.numtasks	1	-	2	-	6	-	18	-	5	-	21
mapreduce.job.maps	2	-	23	-	11	-	35	-	17	-	2

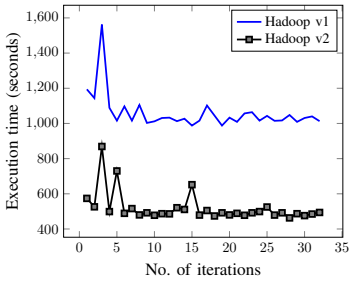
Table I: Default value of parameters and their values tuned by SPSA (the last three parameters are defined for Hadoop v2)



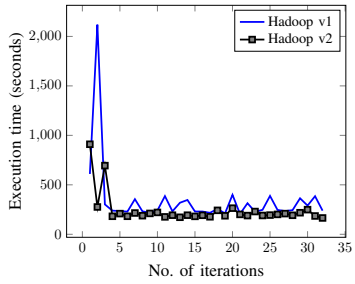
(a) Convergence in Terasort (30GB)



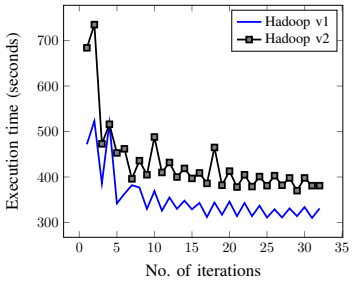
(b) Convergence in Grep (22GB)



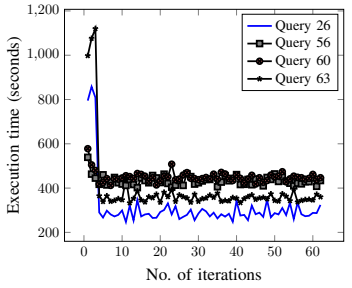
(c) Convergence in Word-Cooccurrence (1GB)



(d) Convergence in Inverted Index (1GB)



(e) Convergence in Bigram (200MB)



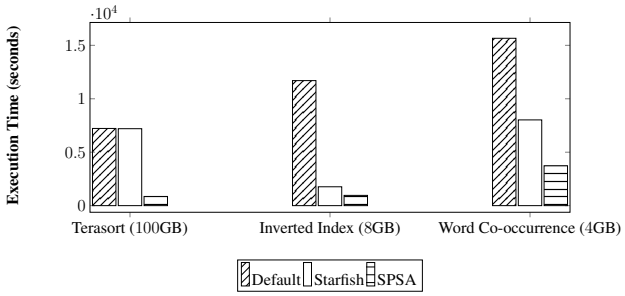
(f) Convergence in Hive Queries (100GB)

Figure 3: Convergence of SPSA for different benchmarks on Hadoop (v1 and v2) and HIVE on MapReduce

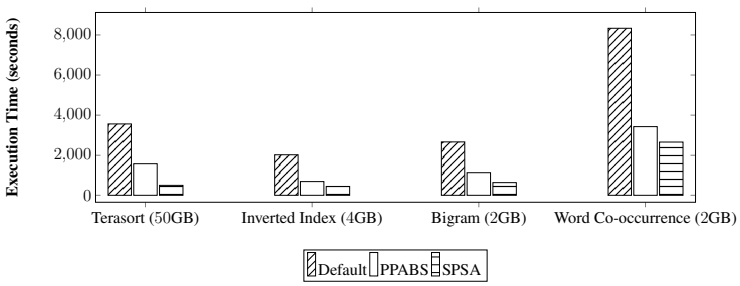
4) Parameters optimized by SPSA are not dependent on the number of nodes. This enables SPSA to be used for optimizing Hadoop on the cloud like Amazon's AWS, Microsoft's Azure, etc.

IV. Conclusions and Future Work

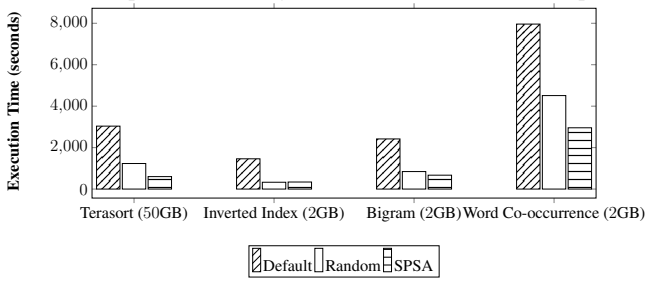
In this paper, we proposed a tuning method based on the simultaneous perturbation stochastic approximation (SPSA) algorithm for Hadoop. The salient features of the SPSA based scheme included its ability to use observations from a real system, its insensitivity to the number of parameters



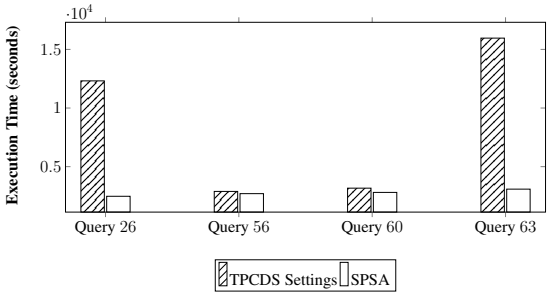
(a) Hadoop Default settings vs Starfish vs SPSA (Hadoop v1)



(b) Hadoop Default settings vs PPABS vs SPSA (Hadoop v2)



(c) Cluster config change, no of nodes from 25 to 12. Performance comparison of Default, Random and SPSA (Average of 5 executions)



(d) HIVE. TPCDS Benchmark. TPCDS Settings vs SPSA (Average of 3 executions)

Figure 4: Benchmark application execution times

and taking the cross-parameter interaction into account via gradients at each point. By optimizing Hadoop parameters using SPSA, we observed reduction in execution times of benchmark applications on a realistic Hadoop cluster. Further, we also showed how our method can be adapted to tune parameters even when the cluster size changes dynamically.

The method we developed can be enhanced to suit more complex hadoop workload scenarios. An immediate area of focus would be to adapt our method to work coherently with Apache Spark [12]. One can also consider reducing the time taken by an iteration in the SPSA learning phase.

V. Acknowledgement

We thank the two anonymous reviewers for their valuable suggestions and comments.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan 2008.

[2] T. White, *Hadoop: The Definitive Guide, Storage and Analysis at Internet Scale, 4th Edition*. O'Reilly Media, March 2015.

[3] K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud." *HotCloud*, vol. 9, p. 12, 2009.

[4] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A Self-tuning System for Big Data Analytics," in *In CIDR*, 2011, pp. 261–272.

[5] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, "MROnline: MapReduce Online Performance Tuning," ser. HPDC '14, 2014, pp. 165–176.

[6] D. Wu and A. S. Gokhale, "A Self-Tuning System based on Application Profiling and Performance Analysis for Optimizing Hadoop MapReduce Cluster Configuration," in *HiPC, 2013*, 2013, pp. 89–98.

[7] J. C. Spall, "Multivariate Stochastic Approximation Using a Simultaneous Perturbation Gradient Approximation," *IEEE Transactions on Automatic Control*, vol. 37, pp. 332–341, 1992.

[8] ——, *Introduction to Stochastic Search and Optimization*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[9] Y. Li and H.-F. Chen, "Robust adaptive pole placement for linear time-varying systems," *IEEE Transactions on Automatic Control*, vol. 41, no. 5, pp. 714–719, 1996.

[10] D. Kushner, H.J.and Clark, *Stochastic Approximation Methods for Constrained and Unconstrained Systems*. Springer-Verlag New York, 1978.

[11] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009.

[12] "Apache Spark," http://spark.apache.org/.